Hindawi Publishing Corporation
EURASIP Journal on Advances in Signal Processing
Volume 2007, Article ID 75373, 15 pages
doi:10.1155/2007/75373

Research Article

SPRINT: A Tool to Generate Concurrent Transaction-Level Models from Sequential Code

Johan Cockx, Kristof Denolf, Bart Vanhoof, and Richard Stahl

Interuniversity Micro Electronics Center (IMEC vzw), Kapeldreef 75, 3001 Leuven, Belgium

Received 1 September 2006; Accepted 23 February 2007

Recommended by Erwin de Kock

A high-level concurrent model such as a SystemC transaction-level model can provide early feedback during the exploration of implementation alternatives for state-of-the-art signal processing applications like video codecs on a multiprocessor platform. However, the creation of such a model starting from sequential code is a time-consuming and error-prone task. It is typically done only once, if at all, for a given design. This lack of exploration of the design space often leads to a suboptimal implementation. To support our systematic C-based design flow, we have developed a tool to generate a concurrent SystemC transaction-level model for user-selected task boundaries. Using this tool, different parallelization alternatives have been evaluated during the design of an MPEG-4 simple profile encoder and an embedded zero-tree coder. Generation plus evaluation of an alternative was possible in less than six minutes. This is fast enough to allow extensive exploration of the design space.

1. INTRODUCTION

Advanced state-of-the-art applications such as multimedia codecs must achieve a high computational power with minimal energy consumption. Given these conflicting constraints, multiprocessor implementations not only deliver the necessary computational power, but also provide the required power efficiency. Multiple processors operating at a lower clock frequency can provide the same performance as a single processor at a higher clock frequency, but with a lower energy consumption [1, 2]. A multiprocessor implementation can be further optimized by selecting a specialized processor for each task, providing a better power-performance trade-off than the single general purpose processor.

An efficient implementation of these applications on such a platform raises two key challenges. First, parallel tasks must be identified and extracted from the sequential reference specification. There must be an excellent match between the extracted tasks and the architecture resources: any significant mismatch results in performance loss, a decrease of resource utilization and reduced energy efficiency of the implementation. Second, the memory and bus/communication network on the platform consume a major part of the energy [3–6], and optimizations reducing this power dissipation are crucial.

However, the task of exploring various program partitions presents one of the major bottlenecks in current design environments. To evaluate a given partition, a concurrent executable model is required. Traditionally, the first concurrent executable model created is a register transfer level HDL model for custom hardware implementation or implementation code for a programmable multiprocessor.

The creation of such a model is clearly too time consuming to allow elaborate exploration of alternatives. Transaction level modeling [7] is an attempt to raise the level of abstraction for concurrent modeling. Although the higher abstraction and higher simulation speed of a transaction level model do facilitate early evaluation of design partitions, the manual creation of such a model is still time consuming and error-prone. In many design cases it is therefore not created at all. As a result, extensive exploration of alternatives is typically not feasible, leading to an inefficient implementation.

To overcome this problem, we have implemented SPRINT, a tool that automatically generates an executable concurrent model in SystemC [8] starting from sequential code and user-defined directives. The SystemC model optionally includes processing delay estimates. SPRINT can automatically derive delay estimates from the source code based on the number and types of operations. Alternatively, delay estimates can be obtained through a theoretical analysis or

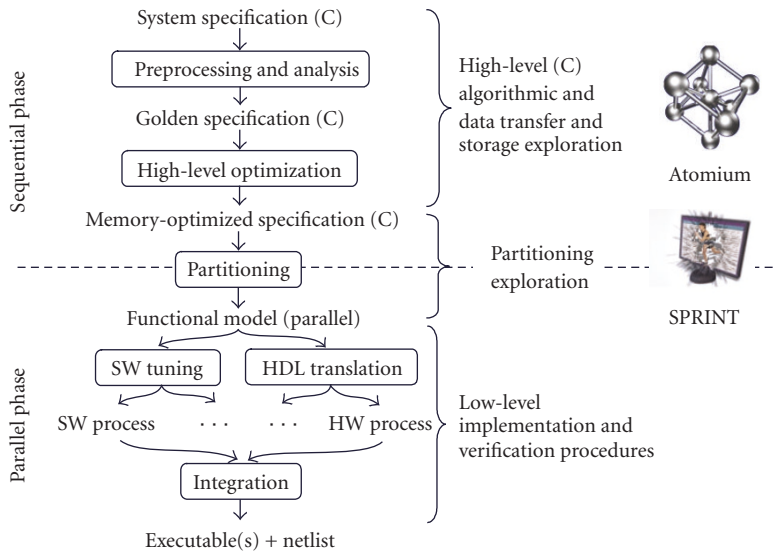

Figure 1: Systematic C-based design flow.

from external estimation tool and can be annotated to the sequential source code. The concurrent timed SystemC model provides valuable performance feedback to the designer in an early phase of the design flow.

SPRINT is part of a systematic C-based design flow targeting the implementation of advanced streaming applications on multiprocessor platforms. It benefits from preprocessing and high-level optimizations applied to the C code before partitioning and support the independent development and testing, potentially by multiple team members in parallel, of parts of the application after partitioning.

SPRINT has been used for several multimedia designs, including an Embedded Zero Tree coder [9] and an MPEG-4 video encoder [10, 11]. With SPRINT, the validation of the concurrent behavior of the complete design was obtained in less than six minutes, including the generation of the model itself. This fast verification path provides the possibility to identify conceptual design errors in an early stage of the design, avoiding expensive design iterations and resulting is a significant speed-up of the design. Moreover, the automated generation of the concurrent model enables the exploration of different parallelization alternatives, ultimately resulting in an improved implementation of the embedded system.

The remainder of this paper is organized as follows. The next section provides an overview of the design flow used for both design cases. The functionality and implementation of the SPRINT tool is detailed Section 3, and experimental results for both design cases are given in Section 4. Section 5 gives a concise overview of related work and discusses the distinguishing features of our approach. The paper is closed with conclusions.

2. SYSTEMATIC DESIGN FLOW

Multimedia application code is usually provided by standardization bodies like MPEG [12]. The provided code typically defines a wide range of options, which are the so-called profiles and levels. The primary goal of this code is to serve as a reference for verification of a realization of particular functionality or profile. The structure of the code is generally not immediately suited for implementation. Direct parallelization of this code for a multiprocessor platform leads to an inefficient software implementation, and direct HDL translation is error-prone and lacks a modular testing environment. A systematic design approach starting with high-level optimizations and supporting parallelization and structured verification is required.

Figure 1 shows the applied design flow, starting from a system specification (typically reference code provided by an algorithm group or standardization body like MPEG) that will gradually be refined into the final implementation: a netlist with a set of executables. Two major phases are present: (i) a sequential phase where the initial specification is preprocessed, analyzed, and high-level optimized and (ii) a parallel phase in which the application is divided into parallel processes and refined to a register transfer level for hardware implementation or implementation code for software implementation. The design flow consists of five logically sequenced steps.

(1) Preprocessing and analysis prunes the reference code to retain only the required functionality for a given application profile. An initial complexity analysis identifies bottlenecks and points out the first candidates for optimization. This step is supported by the Atomium tool suite [13].
(2) High-level optimization reduces the overall complexity with a memory centric focus, as data transfer and storage have a dominant impact on the implementation efficiency of multimedia applications [10, 14]. The code is reorganized into functional modules with localized data processing and limited communication.

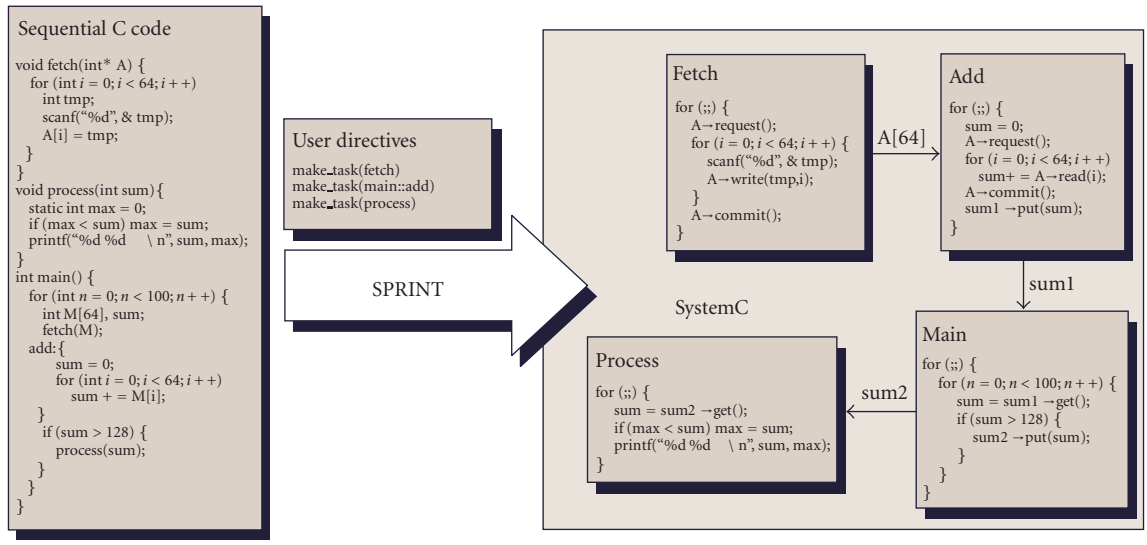

Figure 2: A small example illustrating how SPRINT transforms sequential code to parallel SystemC code with user specified task boundaries. The example includes block FIFO (A[64]) and scalar FIFO (sum1,sum2) channels including a channel (sum2) with a data-dependent communication rate.

This prepares the code for an efficient use of the communication primitives in the next step and facilitates parallelization.

(3) Partitioning splits the code into concurrent tasks and inserts the appropriate communication channels. This step is the focus of this paper. The design is partitioned into concurrent tasks for a combination of reasons, including throughput requirements, power requirements (parallel processors can run at a lower clock rate), design complexity (small tasks are easier to implement, and hardware synthesis tools may yield better results) and the exploitation of heterogeneous processors (tasks can be assigned to the best suited processor, or an efficient custom processor can be designed for a task).

(4) Software tuning and HDL translation refine each task of the partitioned system independently, either by mapping it to a target processor or by writing register transfer level HDL to be synthesized on an FPGA or as custom hardware. In addition, the communication channels between the tasks are mapped on the available communication resources of the target platform.

(5) Integration gradually combines and tests the refined components to obtain a complete working system.

SPRINT supports the designer during the partitioning step by automatically generating a concurrent model and inserting the appropriate communication primitives for designer-specified task boundaries. The generated model can be used to evaluate the partitioning before further refinement of the tasks. Alternatives can be evaluated by changing the task boundary directives and regenerating the model. This way, SPRINT eliminates the necessity to manually identify all accesses to the inputs and outputs of each task and insert the appropriate communication primitives. By automating this time-consuming and error-prone task, SPRINT effectively enables design space exploration. The task code in the generated model can also be used as a behavioral specification for the software refinement or hardware implementation of a task. Finally, the model can be used to generate test stimuli and reference outputs to verify the implementation of each task separately.

3. THE SPRINT TOOL

The SPRINT tool automates the generation of a concurrent model. Based on task boundary directives provided by the designer, SPRINT transforms the sequential C program into a functionally equivalent concurrent SystemC model, consisting of tasks and communication channels. Figure 2 shows the input and output of SPRINT for a small example. This section discusses five key issues in the design of the SPRINT tool: the type of parallelism extracted, the set of communication primitives used, the available mechanisms for timing annotation, the user directives, and the implementation of the tool.

3.1. *Type of parallelism*

Existing parallelizing compilers for multiprocessor platforms target symmetric multiprocessors and exploit data parallelism by executing different iterations of a loop in parallel on different processors. A large body of work exists on the extraction of data parallelism from sequential programs [15–17]. Typically, the techniques used work well on counted loops manipulating arrays in a regular, predictable fashion. Ideally, loop iterations are completely independent. This work is certainly useful and applicable to the design of multimedia applications, as it can be used during task refinement to further split computationally intensive tasks and

thus achieve a better load balance. However, it cannot exploit functional parallelism in which a loop body is split in parts containing different functionality to be executed on different processors.

The tasks generated by SPRINT represent functional parallelism: each task implements a different subset of the statements in the application. By using functional parallelism, heterogeneous processors can be exploited. In case of custom (hardware) processor design, a processor need only implement the functionality of a single task, thus leading to a smaller, more efficient processor that is easier to design and synthesize. Interiteration dependencies do not hinder the extraction of functional parallelism, but naturally lead to a pipelined execution. For this reason, functional parallelism extracted from a loop is often called pipelining. Functional parallelism can often be extracted in spite of the presence of uncounted loops, data-dependent control flow and hard to analyze array accesses. For these reasons, SPRINT only extracts functional parallelism and leaves the extraction of data parallelism, when needed, to other tools to be applied during task refinement.

To distinguish the coarse-grain functional parallelism extracted by SPRINT from fine-grained forms of functional parallelism such as instruction level parallelism (ILP) or software pipelining, we say that SPRINT uses task-level pipelining. The tasks can be seen as pipeline stages communicating through FIFO-like channels.

3.2. Communication channels

Given the designer-specified task boundaries, SPRINT automatically detects and inserts the required communication channels. SPRINT uses a restricted set of communication primitives with well-defined interface functions. The tasks only interact with the channels through these interface functions. The use of interface functions separates computation (in the tasks) from communication (in the channels). Thanks to this separation, tasks and channels can be refined independently.

SPRINT uses two classes of communication channels: unidirectional point-to-point FIFO channels and shared data channels. Each channel has a set of interface functions with a well-defined behavior (Table 1). The implementation of the channel is left unspecified. In fact, different implementations may be used for different target platforms. A FIFO channel can be implemented using any available scheme that is considered appropriate for the target platform, for example a custom hardware block, available synchronization primitives in a shared memory, RTOS primitives, busses or a packet switched network.

The default communication mechanism is a FIFO channel. Two kinds of FIFO channels are used: scalar FIFO and block FIFO. A scalar FIFO uses the traditional put and get interface functions for the communication of scalar tokens. A block FIFO uses a nonstandard set of interface functions to enable zero-copy communication [18] for array tokens. The request function is blocking and waits until a token is available. The read and write functions allow random access to the current token. The commit function releases the current token, either (at the producer side) for transmission to the consumer or (at the consumer side) so that it can be recycled. Together, these functions allow an array token to be constructed in and read from the channel, without copying the array to or from local data. A scalar FIFO can be seen as a special case of a block FIFO, with array size one and using the traditional FIFO interface functions put and get. A put call is equivalent to a request-write-commit sequence, and a get call is equivalent to a request-read-commit sequence.

Table 1: Interface functions for the communication channels inserted by SPRINT.

	Producer functions	Consumer functions
Scalar FIFO	Void put(T)	T get()
Block FIFO	Void request() T read(int) Void write(T, int) Void commit()	void request() T read(int) Void write(T, int) Void commit()
Shared scalar	T read() Void write(T)	T read() Void write(T)
Shared array	T read(int) Void write(T, int)	T read(int) Void write(T, int)

The use of a FIFO-like communication scheme is a logical choice for streaming applications. It naturally leads to a data-flow style implementation [19, 20], with tasks implicitly synchronized through FIFO channels. A task stalls when it needs a token from an empty input queue or when it tries to produce a token on a full output queue. The resulting implementation can be interpreted as a Kahn Process Network [21]. The behavior is deterministic: the outputs depend only on the inputs, and not on the relative order of execution of the tasks. However, correct insertion of FIFO channels such that the resulting Kahn Process Network is functionally equivalent to the original code is a nontrivial task that may require complicated data-flow analysis that is hard or even impossible to automate.

A shared data channel is only used when the communication pattern in the sequential code is such that SPRINT cannot automatically insert a functionally equivalent FIFO. The interface functions for a shared data channel allow random read and write access from multiple tasks without synchronization, but in our design flow, shared data channels are only used for unidirectional point-to-point communication, as a generalized FIFO channel. Under certain conditions, synchronization of accesses to these generalized FIFO channels is redundant due to the presence of plain FIFOs with synchronization (see [22, Chapter 9]). However, to identify shared data as a generalized FIFO, much more sophisticated program analysis techniques are needed than currently implemented in SPRINT. Shared variables are therefore only inserted when explicitly requested by the user. The size of a shared variable must be chosen carefully, taking into account the lifetime of the data it contains and the depth of the FIFO channels, to ensure FIFO-like communication and

thus safeguard deterministic behavior. Functional correctness of the resulting parallel implementation must be verified by simulation of the transaction level model. A concrete example of the use of shared variables is given in Section 4.2.2.

3.3. Timing annotations

By default, SPRINT generates an untimed transaction level model. An untimed model may be sufficient to determine the functional correctness of a task partitioning, but an assessment of the quality of a task partitioning generally requires a timed model. In general, delays occur both in the application code (computation delays) and in the communication channels (computation delays), but the approach used to insert these delays in the SystemC model differs.

To model computation delays, SPRINT can optionally insert wait calls in the generated SystemC code. The argument of the wait call represents the processing time of the adjacent statements on the target platform.

The best way to estimate computation delays depends on the target platform, the available tools and the required accuracy. Tools to profile execution times on specific processors exist [23–25]. In case of custom hardware, an experienced designer may be the only available source of high-level timing estimates. In both cases, the appropriate wait calls can be inserted in the sequential source code. SPRINT will copy such wait calls to the generated SystemC model. The argument of the wait call can be a data dependent expression, thus providing maximum flexibility.

Alternatively, SPRINT can insert symbolic delays during code generation. This method is based on counting the number of operations of different types (read, write, addition, multiplication, ...) found in the C code. The operation count is then multiplied by a symbolic value representing the average execution time for that type of operation on the target processor. An actual value must be provided when the generated SystemC code is executed.

Communication delays are modeled in the SystemC implementation of the channel interface functions (Table 1). By default, a generic implementation is used. This generic implementation models blocking behavior of the FIFO channels, but no communication delays. The generic implementation can easily be replaced by a more detailed, platform specific SystemC model with communication delays.

3.4. User directives

To facilitate the exploration of partitioning alternatives, user directives are not annotated in the sequential source code, but provided in a separate file. This avoids cluttering of the source code and allows a designer to maintain multiple directive files for the same design.

User directives refer to the source code using function names, labels, and variable names. There are three ways to select statements to be moved to a new task:

(i) *make_task(function)* selects all statements in the function body;

(ii) *make_task(function::label)* selects a single (but possibly hierarchical) labeled statement; and

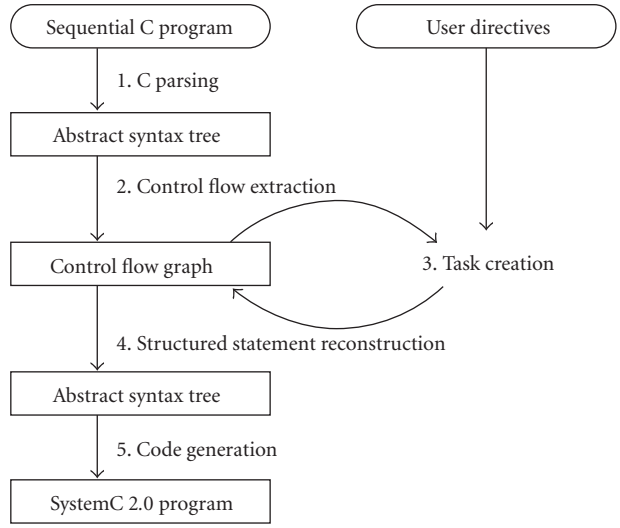

Figure 3: SPRINT consists of five phases transforming a sequential C program into a parallel SystemC program. The user provided input is indicated by rounded boxes and italics.

(iii) *make_task(function::label1-label2)* selects all statements from label1 to label2 inclusive.

Each directive creates a new task. Together with the original "main" task, n directives result in $n + 1$ tasks. By default, FIFO channels are inserted to all intertask communication. Shared data communication channels must be requested explicitly using a shared_variable(var) directive.

3.5. Implementation

As shown in Figure 3, the transformation is done in five phases.

(1) With C parsing, an abstract syntax tree (AST) is generated.

(2) From this AST, a Control Flow Graph (CFG) is extracted for each C function. Nodes correspond to statements, edges to control transfers. Structured statements (if-then-else, for and while loop, block, switch) are flattened. To aid variable liveness analysis, a special clear statement is automatically inserted where local variables in a block statement go out of scope; at that point, the local variable is known to be dead.

(3) During task creation and based on the user directives, the CFG is transformed to model the different concurrent tasks. Channel access statements are inserted as required.

(4) After processing all directives again, an AST is generated from the CFG using structured statement reconstruction techniques.

(5) Finally, the code for the concurrent SystemC model is generated.

Abstract syntax trees, C parsing, control flow extraction and code generation are well-known techniques [26] and are

not further discussed. Structured statement reconstruction is based on natural loops [26] and some heuristics; it is independent of how the input code was structured. A more detailed discussion is outside the scope of this paper.

Task creation

Task creation consists of the following seven steps that are executed for each make_task directive. The result is a newly created task containing the statements selected in the user directive. These seven steps, together with the other phases in Figure 3, are implemented as fully automated routines of SPRINT.

Step 1 (single-entry single-exit check). Control flow edges entering the group of statements selected by the make_task directive are called entry edges; a statement at which such an edge ends is an entry statement. Similarly, control flow edges leaving the selected group of statements are called exit edges; a statement at which such an edge ends is an exit statement. SPRINT checks if there is only one entry and one exit statement for given task. If otherwise, a report is generated, providing details to the designer, who can adjust the task boundaries accordingly.

Step 2 (single parent task check). The function containing the selected statements is called the parent function of the new task. If the parent function is called in two or more parent tasks, channels to or from the new task will not be point-to-point. Such cases are not automatically handled by SPRINT, but a detailed report is generated for the designer.

Step 3 (split channel access check). Channel access statements accessing the same channel must stay within the same task to guarantee the correct order of execution. They must either all stay in the parent task or all move to the new task. Since it is possible that channel access statements have been inserted in the parent task during a previous task creation, the requested task boundaries may cause channel accesses to become distributed over multiple tasks. In such a case, SPRINT does not proceed in task creation process. It generates a report that provides details to the designer, who can adjust the task boundaries accordingly.

Step 4 (identification of communication channels). In this step, SPRINT identifies all base variables of the new task that will be changed into communication channels in the next steps. Due to the existence of pointers in C, it is not always immediately apparent from the source code which variables communicate data between the selected group of statements and the other statements of the parent task.

SPRINT distinguishes a variable reference from a variable access. A statement refers to a variable if the variable's name is mentioned in the statement. It can either directly access (read or write) the variable, or take the address of the variable. A statement accesses a variable if it reads from or writes to the variable. Reads and writes can be direct or indirect (through a pointer). For example, the statement $A[i] = \text{tmp}$ in the sequential C code in Figure 2 refers to the pointer variable A, but (indirectly) writes to the array variable M.

SPRINT identifies communication channels based on variable references, not accesses. This means that communication channels can be identified prior to pointer analysis. A communication channel is inserted for each variable referred to by statements both inside and outside the selected group (Steps 6 and 7). This means that a communication channel always corresponds to a user-named variable, which is called the base variable of the channel. For example, the pointer variable A in Figure 2 is referenced both inside and outside the fetch task. Note that the make_task(fetch) directive creates a task containing the statements in the *body* of the fetch function, and that the declaration of A is in the header of the function, so outside the task. A communication channel is therefore inserted for A, and A is the base variable of the channel.

Note that the need for a communication channel can always be detected by looking at variable references only. In case of communication through an indirectly accessed location, a pointer to that location must have been communicated previously. If the communication of this pointer occurred through a named variable referenced both inside and outside the selected group of statements, that variable flags the presence of a communication channel. Otherwise, the pointer was communicated through an indirectly accessed location, and the same reasoning applies recursively. In the example of Figure 2, data is communicated through the array variable M, but the need for a communication channel is detected by looking at references to A.

In the current implementation, a base variables must be a scalar, an array of scalars, a pointer to a scalar, or a pointer to an array of scalars, where a scalar is a variable that is not an array, struct or pointer. Examples of scalar types in C are int, unsigned char and float. In our experience, this restriction is not a serious limitation for the optimized sequential code for streaming applications to be processed by SPRINT, but it does allow us to use a fast tailored implementation of pointer and liveness analysis (Step 6).

Step 5 (task creation). The new task consists of an infinite loop containing the selected statements. In the parent task, the selected statements are removed and entry edges are set to point to the exit statement.

Step 6 (pointer and liveness analysis). Liveness analysis determines the direction of a FIFO channel to be inserted: an input FIFO is inserted for data that is live at an entry edge, and an output FIFO is inserted for data that is live at an exit edge (Step 7).

Liveness is analyzed for the data to be communicated: for a nonpointer base variable, this is the base variable itself, and for a pointer base variable, this is the set of all locations potentially pointed to. A location is live at an edge in the CFG if the edge is on a path from a statement writing the variable to a statement reading the variable [26]. In Figure 2, liveness is analyzed for M, not for the base variable A. M is live in the body of the for-loop of the fetch function and in the block labeled add.

Modified C code

```
void fetch(int* A) {
  for (int i = 0; i < 64; i++)
    int tmp;
    scanf("%d", &tmp);
    A[i] = tmp;
  }
}
void process(int sum) {
  static int max = 0;
  if (max < sum) max = sum;
  printf("%d %d \n", sum, max);
}
int main() {
  for (int n = 0; n < 100; n++) {
    int M[64], sum, N[64];
    fetch(M);
    fetch(N);
  add:{
      sum = 0;
      for (int i = 0; i < 64; i++)
        sum += M[i]+N[i];
    }
    if (sum > 128) {
      process(sum);
    }
  }
}
```

Figure 4: Modified sequential C code of Figure 2 illustrating the problem of a base variable A that can point to multiple locations.

In the presence of pointers, liveness analysis relies on pointer analysis to identify all accesses (including indirect accesses) to a given location. Note that pointers automatically arise in C when an array is passed as a parameter to a function: C automatically converts the array parameter to a pointer parameter. This is a common situation in C code for multimedia applications. We perform an inter-procedural context-insensitive flow-insensitive Andersen-style [27] pointer analysis for all potential targets of a pointer base variable. Potential targets of pointer base variables are all scalar and array of scalar variables. Heap locations are not analyzed; if a base variable can refer to a heap location, a report is generated the designer, who can take corrective action.

If the base variable of a channel can point to more than one location, and at least one of these locations is an array, an additional context-sensitive analysis is performed to verify that the target locations are never live at the same time. This is a necessary condition for the use of zero-copy channels (shared array and block FIFO, see Section 3.2). To avoid the overhead of a general context-sensitive analysis, this analysis is only done for the potential targets of a pointer base variable. If the condition is not fulfilled, a report is generated providing details for the designer, who can than adjust the code appropriately, taking design constraints into account.

To illustrate the problem of a base variable pointing to multiple locations, assume that the sequential code in Figure 2 is modified such that it contains a second call to the fetch function, as in Figure 4. Now A can point to M or N, and M and N are live at the same time. Since the block FIFO channel A cannot contain both M and N at the same time, M and N would have to be communicated one by one. Since a task can only access one token at a time in a given block FIFO channel, the first array would then have to be copied out of the channel into a temporary buffer before the second array could be accessed. In general, the copying cancels the effect of zero-copy communication and may ruin high-level memory optimizations applied during the sequential phase of our systematic design flow (Section 2). For this reason, we prefer to ask the designer to take appropriate action.

Step 7 (channel insertion). For each base variable, a channel is inserted. The kind of channel depends on user directives (for shared data) and on the type of the communicated data.

(1) A shared data channel is inserted only if the base variable is marked as shared by a user directive. If the base variable is an array, or if it is a pointer and at least one location pointed to is an array, a shared array channel is inserted. Otherwise, a shared scalar is inserted. All accesses to the channel data, including indirect accesses, are replaced by read() and write() operations on the channel. The simple example in Figure 2 does not contain shared data channels, but three examples of shared data channels in an MPEG-4 encoder will be discussed further (Section 4.2).

(2) Otherwise, if the base variable is an array, or if it is a pointer and at least one location pointed to is an array, a block FIFO is inserted. To avoid array copying, the data to be communicated will be constructed directly in the array. This means that every access to the array is replaced by a read() or write() operations on the block FIFO. If a pointer dereference may access more than one location, all the locations it may access must be part of the same block FIFO channel, so that the access can be replaced by a single read() or write() operations. In addition, a request() call is inserted before the first access and a commit() call is inserted after the last access, both in the new task and in the parent task. To maximize parallelism and decrease the probability of split channel access problems (Step 3), code motion is used to move the request() and commit() calls as close as possible to the data accesses.

In the example of Figure 2, $A[i]$ = tmp is replaced by A->write(tmp, i) and sum += $M[i]$ is replaced by sum += A->read(i) during processing of the make_task(fetch) directive. Due to code motion, the A->request() and A->commit() calls in the parent task are moved into the block labeled add, just before and after the for-loop, respectively. During processing of the make_task(main::add) directive, the A->request() and A->commit() calls are moved to the add task, together with the other statements in the block. Without code motion, these calls would have stayed outside the block, and the split channel access check (Step 3) would have failed.

The direction of the FIFO depends on the results of liveness analysis (Step 6). If channel data is only live at an entry edge of the new task, a channel is created from the parent task to the new task, and vice versa. If channel data can be live both at an entry edge and at an exit edge, a report is generated to the designer, who can take corrective action. An automatic solution would involve copying of arrays, which is to be avoided.

(3) Otherwise, the base variable is either a scalar or a pointer to scalar locations, and a scalar FIFO is inserted. As for block FIFOs, the direction of the FIFO depends on the result of liveness analysis. A put() call is inserted in the producer task after the last access, and a get() call is inserted in

Table 2: Experimental results for the EZT coder (801 lines of sequential C code).

Configuration	No. of tasks	Target	SPRINT (s)	SystemC compilation (s)	Simulation speed (s/image 256×256)	No. of cycles per frame	Speed-up
seq	1	SW	0.3	3.0	2.22	170.6 M	1.0x
par-a	3	SW	1.6	3.6	3.12	141.6 M	1.2x
par-b	7	SW	3.4	4.0	11.11	43.6 M	3.9x
par-c	5	SW	2.5	3.9	8.33	44.0 M	3.9x

the consumer task before the first access. To maximize parallelism and decrease the probability of split channel access problems (Step 3), code motion is used to move the put() and get() calls as close as possible to the data accesses. Examples of scalar FIFO channels in Figure 2 are sum1 and sum2.

4. EXPERIMENTAL RESULTS

To asses the usability of SPRINT, the following questions must be answered.

(1) Can SPRINT effectively parallelize real applications on realistic platforms?
(2) Is the evaluation of a partitioning, including model generation, compilation and simulation, fast enough to support interactive exploration of the design space?

To obtain answers to these questions, we have applied SPRINT on two multimedia designs: an Embedded Zero Tree Coder (about 800 lines of sequential C code) and an MPEG-4 Simple Profile Video Encoder (about 10000 lines of sequential C code). A description of these designs and how they were parallelized is given below. The results are summarized in Tables 2 and 3. All results were obtained on a Pentium 4 3.20 GHz with 1 GB RAM running RedHat Linux 2.4.21 with gcc 3.2.3, SystemC 2.1 and SPRINT 2.8.13. Reported execution times are elapsed times on an otherwise unloaded processor.

4.1. Embedded zero-tree coder

Embedded Zero Tree (EZT) coding is a recursive compression scheme for wavelet coefficients [28, 29]. It is used for image compression in MPEG-4 standard for Visual Texture Coding [12] for 3D modeling. In contrast to the well-known MPEG-4 video coding, this is an image coding application.

The sequential C code used in this test case has been taken from the design data of the OZONE chip [9]. The preprocessing and high-level optimization phases of our systematic design flow had already been applied to this code. During optimization, the recursive core of the reference code was replaced by two iterative passes: a code definition pass and a code production pass. This transformation facilitates memory optimizations and at the same time enables a parallel implementation of the two passes. For hardware implementation, the code was then partitioned in three pipelined tasks: code definition, code production and arithmetic coding. A detailed explanation of the EZT algorithm [28, 29] and the optimizations applied for hardware implementation [9] is beyond the scope of this paper.

We have used SPRINT to explore a software mapping for this design on a multiprocessor platform. For timing, we used SPRINT's automatic insertion of wait statements, and assumed that every operation takes one clock cycle. The generic implementation of the communication channels, with zero communication delay, was used.

Three partitioning configurations have been evaluated. In configuration par-a, SPRINT creates the same three tasks that were used in the hardware implementation. Simulation results (Table 2) for a 256×256 pixels version of the well-known Lena image show that a speed-up factor of only 1.2x is achieved. An examination of the task activity diagram reveals that the bottleneck is the code definition task. This task mainly performs bit manipulations that are comparatively slower in software than in hardware. In configuration par-b, the code definition task was therefore further parallelized by splitting of four additional tasks. The resulting task diagram is shown in Figure 5. Assuming one processor per task, a speed-up of 3.9x can now be achieved on 7 processors. The task activity diagram for this configuration (Figure 6) shows that the processors for the code definition task and the arithmetic coder task are barely loaded. In configuration par-c, the task boundaries are therefore changed so that the code definition functionality is in the same task as the compute code3 functionality, and the arithmetic coder functionality is in the same task as the code production functionality. This configuration achieves practically the same speed up on just five processors.

The execution times for SPRINT model generation, SystemC compilation and the simulation speed are listed in Table 2. A new configuration for this 800 line example can be evaluated in less than one minute. This is more than fast enough to allow interactive exploration.

4.2. MPEG-4 simple profile video encoder

SPRINT has been applied during the design of a high-throughput and low-power custom chip [30] implementation of an MPEG-4 Simple Profile video encoder. The same design has also been partitioned for a software implementation on a dual core platform with DMA. We use this design not only to provide experimental results, but also to illustrate the use of shared data channels in SPRINT.

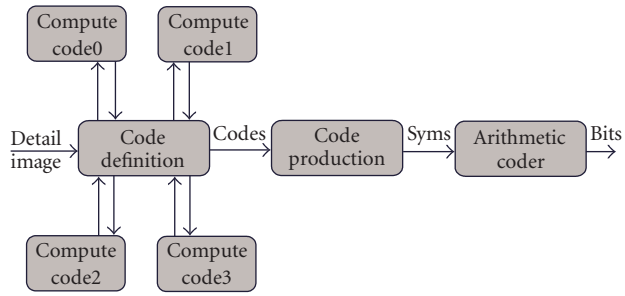

Figure 5: Task diagram for configuration par-b of the EZT coder. Thin lines are scalar FIFOs and thick lines are block FIFOs.

Table 3: Experimental results for different parallelizations of the MPEG-4 simple profile encoder.

Configuration	No. of tasks	Target	SPRINT (s)	SystemC compilation (s)	Simulation speed (QCIF frames/s)	No. of cycles per frame	Speed-up
seq-hw	1	HW	8.3	5.9	24.1	512 K	1.0x
par-hw	9	HW	122.8	11.2	16.4	238 K	2.2x
seq-sw	1	SW	9.2	7.5	20.8	23.9 M	1.0x
par-a	9	SW	126.1	14.4	1.40	13.3 M	1.8x
par-b	4	SW	77.6	10.5	1.25	13.4 M	1.8x

4.2.1. Functionality, preprocessing, and high-level optimizations

The MPEG-4 part 2 video codec [31] belongs to the class of lossy hybrid video compression algorithms [32]. Figure 7 gives a high-level view of the encoder. A frame is divided in macroblocks, each containing 6 blocks of 8×8 pixels: 4 luminance and 2 chrominance blocks. The motion estimation (ME) exploits the temporal redundancy by searching for the best match for each new input block in the previously reconstructed frame. The motion vectors define this relative position. The remaining error information after motion compensation (MC) is decorrelated spatially using a DCT transform and is then quantized (Q). The inverse operations Q^{-1} and IDCT (completing the texture coding chain) and the motion compensation reconstruct the frame as generated at the decoder side. Finally, the motion vectors and quantized DCT coefficients are variable length encoded. Completed with video header information, they are structured in packets in the output buffer. A rate control algorithm sets the quantization degree to achieve a specified average bit rate and to avoid over or under flow of this buffer.

A discussion of all the optimizations applied during the sequential phase of our systematic design flow (Section 2) is beyond the scope of this paper, but one particular optimization is relevant to illustrate the use of shared data channels in SPRINT. The original specification contains two frame buffers for the reconstructed frame (Figure 7), to allow motion estimation and motion compensation to access the current frame while the next frame is constructed. However, as frame buffers are large and costly, only a single frame buffer is retained. As long as motion estimation and compensation process the frame in a row-by-row order and the update of the frame happens in the same order, it is sufficient to prefetch the pixels needed for motion estimation and compensation and store them in a separate, smaller buffer before they are overwritten. As a high-level optimization, the sequential C code has been modified to implement this prefetching and remove the double frame buffer.

4.2.2. Partitioning for hardware mapping

To achieve the high throughput requirement (47520 macroblocks/s, equivalent to 18.25 megapixels/s) at a low-power consumption [30], a video pipeline architecture is envisaged exploiting the inherent functional parallelism of the encoding algorithm. Every functional unit becomes a separate stage of the pipeline, and is implemented as a task specific processor, except for the rate control task, which is implemented as a software task to be executed on an embedded processor. Each channel is implemented as an on-chip two-port memory, except for the reconstructed frame buffer that is assigned to an external memory.

Before moving to the (work intensive) register transfer level implementation of the processors, a SystemC transaction level model was generated by SPRINT. The partitioning used for this design consists of 9 tasks (Figure 8). To model timing, wait statements were manually added to the code of each task by an experienced designer, based on the expected number of memory accesses and processing cycles for the corresponding processor. The model was used to verify functional correctness, to assess the achieved parallelism and to generate a test bench per task, allowing verification of the register transfer level implementation of each processor.

The shared data channels require special attention. For efficiency reasons, three channels have been implemented as shared data channels: reconstructed frame, search area and

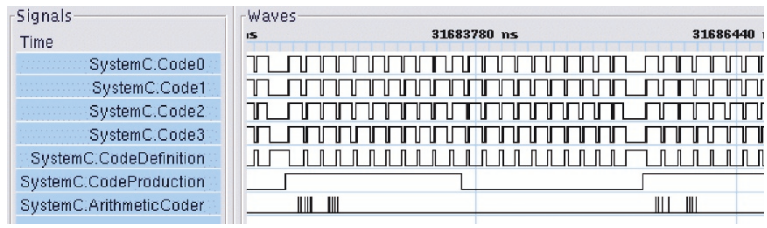

Figure 6: Extract of the task activity diagram generated during simulation of the EZT coder in configuration par-b.

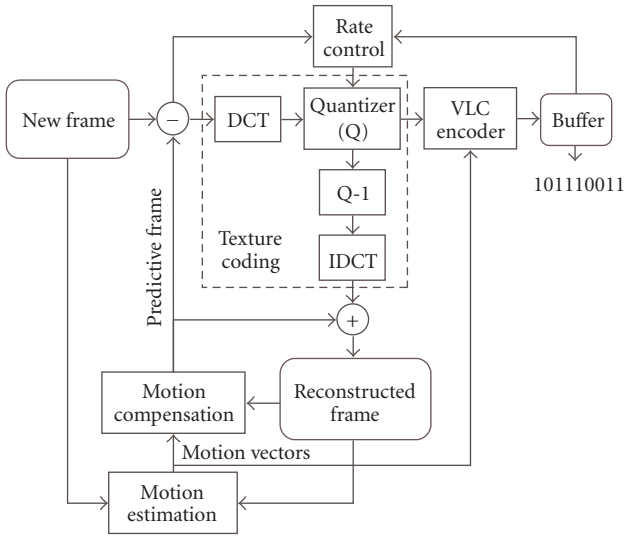

Figure 7: Functional view of an MPEG-4 video encoder.

YUV buffer. Shared data channels provide no synchronization. Under certain conditions, however, the synchronization of shared data accesses is redundant due to the presence of FIFO channels between the same tasks (see [22, Chapter 9]). SPRINT cannot verify that these conditions are fulfilled, but the designer can use the generated concurrent model to check the correct synchronization through simulation.

Consider for example the search area channel between the copy controller and the motion estimation (Figure 8). When motion estimation advances to the next block, the new search area overlaps the previous search area (Figure 9). For efficiency reasons, the overlapping part of the search area is reused, and only the nonoverlapping part is written by the copy controller. This access scheme was introduced as a high-level memory optimization during the sequential phase of the design flow. As a regular FIFO does not support data reuse, the search area communication channel has been implemented as a circular buffer in a shared data channel.

In the sequential code, the copy controller and the motion estimation execute in turn. In the concurrent model, however, this is no longer true, the second execution of the copy controller can start during or even before the first execution of the motion estimation. This means that the size of the search area buffer must be increased so that it can hold the data produced by two or more executions of the copy controller, as well as reused data. The required size is limited by the depth of the scalar FIFO from copy controller to motion estimation. Every execution of the copy controller also produces a token on this FIFO. When the FIFO is full, the copy controller will block until the motion estimation has executed at least once. If the depth of the FIFO is N, the copy controller can be a most N executions ahead of the motion estimation.

A similar reasoning can be made about the synchronization of accesses to the YUV buffer and the reconstructed frame. In these two cases, there is no direct FIFO between the tasks accessing the shared data, but there is in both cases an indirect path consisting of FIFOs only. Such an indirect path can also provide sufficient synchronization; the maximum lag between the producing and consuming tasks is related to the sum of the depths of the FIFOs on the path. In general, a necessary (but not sufficient) condition for correct synchronization of shared data accesses is that there is a direct or indirect path consisting of FIFO channels between the tasks accessing the shared data. This condition is checked by SPRINT.

The amount of data produced by the copy controller is variable in our design. Usually, three macroblocks are produced, but at the edges of the frame, there are more. At a certain moment during the design process, this was not taken into account. The problem could not be detected by executing the sequential code. Without the concurrent model generated by SPRINT, this synchronization problem would only have been detected during integration of the register transfer level code of the processors, and a fix would have been much more costly.

4.2.3. *Partitioning for software mapping*

As an additional experiment, we have used SPRINT to construct a partitioning for a dual core processor with a DMA controller. For timing, we used the same approach as for the EZT design: automatic insertion of wait statements and one clock cycle per operation. If the same partitioning in 9 tasks as for the hardware implementation (Figure 8) is used, the activity diagram reveals that the motion estimation task is the bottleneck and consumes nearly as much processing time as all other tasks together. It was therefore decided to assign motion estimation to the first processor and all other tasks except the input controller and the copy controller to the second processor. Input controller and copy controller were assigned to the DMA controller. To reduce

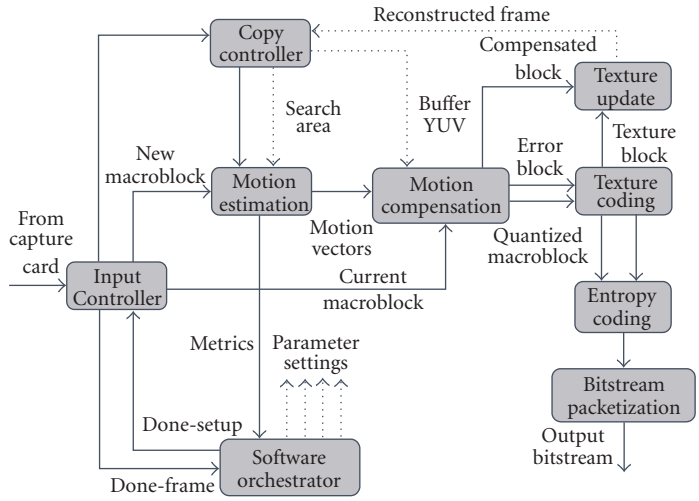

Figure 8: Task diagram for the par-hw and par-a configurations of the MPEG-4 simple profile encoder. Thin lines are scalar FIFOs, thick lines are block FIFOs, and dotted lines are shared variables. To avoid cluttering the diagram, parameter setting channels are shown only partially.

Figure 9: Data reuse in the overlapping region of the search area for motion estimation.

context switching overhead on the second processor, the motion compensation, texture update, texture coding, entropy coding and bit-stream packetization functions were implemented as a single task. Merging the software orchestrater with this task proved undesirable, because it introduced a sequence constraint that reduced the effective parallelism. The final configuration contains four tasks: a motion estimation task, a DMA task implementing input controller and copy controller, a software orchestrater task and a task implementing the remaining functional blocks.

4.2.4. Results

Results for both hardware and software partitioning are summarized in Table 3. The seq-hw and seq-sw configurations correspond to a sequential execution with timing for the hardware and software implementations, respectively, and are used as a reference point to evaluate the achieved speed-up. Simulation speed was measured using the Foreman QCIF test sequence containing 299 frames.

The processing speed observed during simulation of the par-hw configuration (238 Kcycles/frame) is close to the processing speed measured on the hardware implementation [10] (3.2 MHz for 15 frames/s QCIF, corresponding to 213 Kcycles/frame), indicating that the timing estimates used were realistic. The speed-up obtained for this configuration is only 2.2x for nine processors, but this is not a problem as the target of the partitioning was not to obtain the highest possible throughput, but low-power consumption and an efficient design process. The partitioning used results in relatively simple processors for which an energy-efficient implementation is possible. These processors consume practically no power when idle. Also, smaller processors are easier to describe in VHDL and are handled more efficiently by synthesis tools. In addition and thanks to the separation of computation and communication, multiple designers can work on the refinement of these processors in parallel.

Starting from 9588 lines of sequential C code and user-defined task boundaries for up to nine tasks, SPRINT can generate a concurrent SystemC model in about two minutes. The simulation of a complete QCIF sequence (299 frames) for the hardware partitioning (par-hw configuration) is finished in less than 20 seconds. In comparison, simulation with the Modelsim VHDL simulator, only possible after completion of the HDL design of the encoder system, requires more than 6 hours for the same QCIF sequence. Simulation speed for the software configurations is an order of magnitude slower than for the hardware configurations. This is due to the more detailed timing annotations used in the software configurations. Even in the slowest configuration (par-b), a simulation of the complete test sequence takes only 4 minutes.

The fast generation of a high-level concurrent model combined with the gain in simulation speed allows the exploration of different parallelization alternatives. Moreover, due to the gain in simulation performance (about three orders of magnitude), an extensive test set can be used to assess the system performance early in the design cycle before even the architecture mapping or hardware design has started. For all configurations, the total time needed to generate a new concurrent model and evaluate it using the 299-frames Foreman

Table 4: A comparison of SPRINT with related approaches.

	Input	Partitioning	Data-flow analysis	Output
COMPAAN	ANLP	Automatic	Static, array	YAPI KPN
FP-MAP	C	Automatic	Dynamic	None
DSWP	C	Automatic	Static, recursive data structures	Itanium object code
SPRINT	C	User-defined	Static, scalar	SystemC TLM

QCIF test sequence is less than six minutes. We believe that this is more than fast enough to allow interactive exploration of the design space.

5. RELATED WORK

Parallel computing and parallelization are wide research areas that have been studied in a large number of projects. Due to this broad scope, we only list the most closely related approaches that have focused on the task-level pipelining.

Starting from an initial sequential implementation (of a given application, typically written in C), two general approaches to multiprocessor partitioning are manual refinement and automated parallelization. The manual approach to embedded system mapping is typically based on a systematic methodology [33]. This, however, is a difficult and error-prone process, as shown in [34]. The final solution can be very optimal for given platform, but is typically very platform-specific, hard to adapt and maintain. Moreover, a complete rewriting or a new solution has to be developed for other platforms.

Another manual approach to multiprocessor partitioning is the use of higher level programming languages that express parallelism in an explicit form, for example, parallel languages with support for task-level pipelining as ZPL [35], or more specialized languages, for example, streaming languages as StreamC [36] or KernelC [37]. Use of such languages, accompanied with an effective compiler and an appropriate back-end for given platform, can considerably simplify the exploration and mapping process. Even though this approach can lead to solutions with good performance (as reported in [34] for ZPL), it still requires manual code rewriting from the initial implementation to the chosen programming language, which is a nontrivial task.

Automated parallelization in general requires either advanced compilers [15–17, 38] or source-to-source transformation tools [39]. Even though the topic of automated parallelization has been researched for decades, the problem of effective code parallelization seems to persist. Furthermore, only very few approaches, such as IBM X1 HPF [38], have implemented support for task-level pipelining, while the most advanced compilers, such as SUIF [15], have focused on advanced program analysis and transformations for data parallelism.

The IBM X1 HPF compiler [38] and its derivatives are to our knowledge the only compilers that automatically perform functional pipelining without user annotations to the input code (HPF). As reported in [34], the compiler performs well, yet only on a limited number of kernels. It does not pipeline non-perfectly nested loops, or more complex control-flow that could still be pipelined with a minor user intervention. Similar restrictions/results seem to be very common also for the other automated approaches to parallelization.

The approaches that are most closely related to SPRINT are COMPAAN, FP-MAP, and DSWP. Like SPRINT, they use task-level pipelining to parallelize embedded applications. The key features of these approaches are compared in Table 4.

COMPAAN [40, 41] automatically creates a task (or process) for each function call in an affine nested loop program (ANLP) written in MATLAB syntax. Using the results of an exact array data-flow analysis, array inputs and outputs are converted to special optimized FIFOs that can reorder and/or duplicate tokens as necessary. The output representation generated by COMPAAN is a Kahn Process Network (KPN). One of the limitations of this approach is that the structure of the ANLP determines the task boundaries; to obtain a different partitioning, the source code must be edited. Task boundaries cannot be located in code executed under nonlinear or data-dependent conditions. This restricts the usability of COMPAAN to only ANLP input, which reduces its practicality by processing only to specific, well-formed loops.

FP-MAP [42] accepts arbitrary C code and is not limited to ANLP as COMPAAN. It automatically distributes the statements of a given loop nest over a given number of processors, minimizing the initiation interval of the pipeline. The distribution algorithm relies on estimates of execution times and branch execution probabilities obtained via profiling. Execution times are assumed to be independent of the processor, and the cost and feasibility of the resulting communication on the target platform is not taken into account. The data-flow analysis used in FP-MAP is dynamic: data dependences are registered during execution of the C code for a given set of inputs. This means that FP-MAP can analyze data flow for arbitrary control flow including data-dependent conditions and for arbitrary index expressions. FP-MAP does not automatically insert communication channels or generate code; the result is an assignment of statements to processors. Thus, it is an interesting approach to task-level pipelining, yet it is not a practical tool usable for mapping real-life multimedia programs to multiprocessor embedded systems.

DSWP [43] is another automated compiler-based approach to task-level pipelining. Its focus is, similarly to ours, on the outer loops with many iterations. In particular, the DSWP technique focuses on recursive data structures in such loop constructs. The technique is based on identification of

Strongly-Connected Components (SCC) in the dependence graph of the input program. Such an SCC must not be split to ensure unidirectional flow of dependences between the resulting tasks. Similar to FP-MAP, a load balancing heuristic is used to distribute the SCCs over the pipeline stages (tasks). DSWP uses the so-called synchronization array for communication among threads [44]. The synchronization array is dedicated hardware that implements a simple FIFO channel. The approach leverages prior research on instruction level parallelism (ILP) by utilizing an existing compilation framework, the impact compiler. Thus, it has a back-end that is currently targeting the itanium processors. Some of the disadvantages of this approach are missing communication optimization, no support for source-to-source transformation, and no possibility of user interaction that we believe is needed for exploration of the solution space for optimized application mapping.

SPRINT provides a unique combination of features that is especially suited to explore the partitioning of an embedded streaming application. It accepts any ANSI-C code. Task boundaries are defined by the designer, which is an advantage given the less than ideal results obtained with existing automatic parallelization tools. The required communication channels are automatically detected and inserted using well-known, robust, and fast scalar data-flow analysis techniques. These techniques are sufficiently powerful to handle complex real-life applications with nonlinear index expressions and data-dependent conditions. In our design flow, sequential optimization steps precede the partitioning, which facilitates the data-flow analysis. If desired, a more advanced data-flow analysis can be used instead. Finally, SPRINT generates a concurrent transaction level model that can be used to interactively explore alternative solutions and rapidly evaluate their performance. The generated model is also suitable for further automated of manual refinement. To the best of our knowledge, SPRINT is the first tool to generate a concurrent transaction level model from sequential ANSI-C code.

6. CONCLUSIONS

To support our systematic C-based design flow for advanced multimedia applications with stringent performance and power requirements, we have developed a tool called SPRINT that generates a concurrent SystemC transaction level model starting from sequential C code and user-selected task boundaries. This tool automates the time-consuming and error-prone task of manually creating such a model, thus encouraging the designer to extensively explore alternative partitionings and avoid the premature selection of a suboptimal implementation. To the best of our knowledge, SPRINT is the first tool to generate a concurrent transaction level model from sequential C code.

To match the characteristics of the target applications and enable effective exploitation of heterogeneous target platforms, SPRINT uses task level pipelining and a carefully selected set of FIFO-like communication channels. The implementation can handle full ANSI-C code and relies on selective and tuned program analysis techniques to achieve a high performance.

The effectiveness of the SPRINT tool has been experimentally verified by applying it to two designs: an Embedded Zero Tree Coder (800 lines of sequential C code) and an MPEG-4 Simple Profile Encoder (10000 lines of sequential C code). For both designs, multiple concurrent models for different realistic target platforms, including both hardware and software platforms, have been generated and evaluated. In all cases tried, the model could be generated, compiled and evaluated in less then six minutes. This is fast enough to allow extensive exploration of the design space.

ACKNOWLEDGMENT

The authors would like to thank Henk Corporaal for the insightful discussions and his valued contribution to this paper.

REFERENCES

[1] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-power digital design," in *Proceedings of the IEEE Symposium on Low Power Electronics*, pp. 8–11, San Diego, Calif, USA, October 1994.

[2] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein, "Scaling, power, and the future of CMOS," in *Proceedings of IEEE International Electron Devices Meeting (IEDM '05)*, p. 7, Washington, DC, USA, December 2005.

[3] H. De Man, F. Catthoor, G. Goossens, et al., "Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 319–335, 1990, special issue on the future of computer-aided design.

[4] T. H. Meng, B. M. Gordon, E. K. Tsern, and A. C. Hung, "Portable video-on-demand in wireless communication," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 659–680, 1995, special issue on low power electronics.

[5] A. Lambrechts, P. Raghavan, A. Leroy, et al., "Power breakdown analysis for a heterogeneous NoC platform running a video application," in *Proceedings of the 16th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '05)*, pp. 179–184, Samos, Greece, July 2005.

[6] M. A. Viredaz and D. A. Wallach, "Power evaluation of a handheld computer," *IEEE Micro*, vol. 23, no. 1, pp. 66–74, 2003.

[7] L. Cai and D. Gajski, "Transaction level modeling in system level design," Tech. Rep. 03-10, CECS, University of California, Irvine, Calif, USA, 2003.

[8] http://www.systemc.org/.

[9] B. Vanhoof, M. Peon, G. Lafruit, J. Bormans, M. Engels, and I. Bolsens, "A scalable architecture for MPEG-4 embedded zero tree coding," in *Proceedings of the 21st IEEE Annual Custom Integrated Circuits Conference (CICC '99)*, pp. 65–68, San Diego, Calif, USA, May 1999.

[10] K. Denolf, C. De Vleeschouwer, R. Turney, G. Lafruit, and J. Bormans, "Memory centric design of an MPEG-4 video encoder," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 5, pp. 609–619, 2005.

[11] K. Denolf, et al., "A systematic design of an MPEG-4 video encoder and decoder for FPGAs," in *Proceedings of the Global Signal Processing Expo and Conference (GSPx '04)*, Santa Clara, Calif, USA, September 2004.

[12] http://www.mpeg.org/.
[13] Atomium, http://www.imec.be/atomium.
[14] F. Catthoor, E. de Greef, and S. Suytack, *Custom Memory Management Methodology*, Kluwer Academic Publishers, Boston, Mass, USA, 1998.
[15] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, et al., "Maximizing multiprocessor performance with the SUIF compiler," *Computer*, vol. 29, no. 12, pp. 84–89, 1996.
[16] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, no. 12, pp. 1184–1201, 1986, special issue.
[17] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452–471, 1991.
[18] D. R. Cheriton, "The V distributed system," *Communications of the ACM*, vol. 31, no. 3, pp. 314–333, 1988.
[19] E. A. de Kock, G. Essink, W. J. M. Smits, et al., "YAPI: application modeling for signal processing systems," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 402–405, Los Angeles, Calif, USA, June 2000.
[20] W. A. Najjar, E. A. Lee, and G. R. Gao, "Advances in the dataflow computational model," *Parallel Computing*, vol. 25, no. 13-14, pp. 1907–1929, 1999.
[21] G. Kahn, "The semantics of simple language for parallel programming," in *Proceedings of the IFIP Congress on Information Processing*, J. L. Rosenfeld, Ed., pp. 471–475, North-Holland, Stockholm, Sweden, August 1974.
[22] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker, New York, NY, USA, 2000.
[23] R. Eigenmann and P. McClaughry, "Practical tools for optimizing parallel programs," in *Proceedings of the SCS Multiconference*, Arlington, Va, USA, March-April 1993.
[24] I. Park, M. J. Voss, B. Armstrong, and R. Eigenmann, "Interactive compilation and performance analysis with ursa minor," in *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, pp. 163–176, Minneapolis, Minn, USA, August 1997.
[25] D. A. Reed, P. C. Roth, R. A. Aydt, et al., "Scalable performance analysis: the Pablo performance analysis environment," in *Proceedings of the Scalable Parallel Libraries Conference*, pp. 104–113, Mississippi State, Miss, USA, October 1993.
[26] A. Aho, R. Sethi, and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass, USA, 1986.
[27] L. O. Andersen, *Program analysis and specialization for the C programming language*, Ph.D. thesis, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, May 1994.
[28] J. M. Shapiro, "Embedded image coding using zero trees of wavelet coefficients," *IEEE Transactions on Signal Processing*, vol. 41, no. 12, pp. 3445–3462, 1993.
[29] J. M. Shapiro, "A fast technique for identifying zero trees in the EZW algorithm," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '96)*, vol. 3, pp. 1455–1458, Atlanta, Ga, USA, May 1996.
[30] K. Denolf, A. Chirila-Rus, and D. Verkest, "Low-power MPEG-4 video encoder design," in *Proceedings of the IEEE Workshop on Signal Processing Systems Design and Implementation (SIPS '05)*, pp. 284–289, Athens, Greece, November 2005.
[31] "Information technology-Generic coding of audio-visual objects—part 2: visual," ISO/IEC 14496-2:2004, June 2004.
[32] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, Kluwer Academic Publishers, Boston, Mass, USA, 1997.
[33] A. C. Downton, R. W. S. Tregidgo, and A. Cuhadar, "Top-down structured parallelization of embedded image processing applications," *IEE Proceedings-Vision, Image and Signal Processing*, vol. 141, no. 6, pp. 431–437, 1994.
[34] E. C. Lewis and L. Snyder, "Pipelining wavefront computations: experiences and performance," in *Proceedings of the 5th IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '00)*, pp. 261–268, Cancun, Mexico, May 2000.
[35] L. Snyder, *A Programmer's Guide to ZPL*, MIT Press, Cambridge, Mass, USA, 1999.
[36] A. Das, W. J. Dally, and P. Mattson, "Compiling for stream processing," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06)*, pp. 33–42, Seattle, Wash, USA, September 2006.
[37] P. Mattson, W. J. Dally, S. Rixner, U. J. Kapasi, and J. D. Owens, "Communication scheduling," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, pp. 82–92, Cambridge, Mass, USA, November 2000.
[38] M. Gupta, S. Midkiff, E. Schonberg, et al., "HPF compiler for the IBM SP2," in *Proceedings of the ACM/IEEE Supercomputing Conference*, vol. 2, pp. 1944–1984, San Diego, Calif, USA, December 1995.
[39] D. B. Loveman, "Program improvement by source-to-source transformation," *Journal of the Association for Computing Machinery*, vol. 24, no. 1, pp. 121–145, 1977.
[40] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "Compaan: deriving process networks from Matlab for embedded signal processing architectures," in *Proceedings of the International Conference on Design Automation and Test in Europe*, Paris, France, February 2004.
[41] A. Turjan, B. Kienhuis, and E. Deprettere, "Translating affine nested-loop programs to process networks," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 220–229, Washington, DC, USA, September 2004.
[42] I. Karkowski and H. Corporaal, "FP-map - an approach to the functional pipelining of embedded programs," in *Proceedings of the 4th International Conference on High Performance Computing (HiPC '97)*, pp. 415–420, Bangalore, India, December 1997.
[43] G. Ottoni, R. Rangan, A. Stoler, M. J. Bridges, and D. I. August, "From sequential programs to concurrent threads," *IEEE Computer Architecture Letters*, vol. 5, no. 1, pp. 6–9, 2006.
[44] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*, pp. 177–188, Antibes Juan-les-Pins, France, September-October 2004.

Johan Cockx received his degree in electrical engineering from the Katholieke Universiteit, Leuven, Belgium, in 1983. From 1983 to 1985, he was a member of the CAD Research Group at the ESAT Laboratory of that university, working on modular circuit simulation. From 1986 to 1996, he worked for Silvar-Lisco, later renamed EDC, on a wide range of electronic design tools

including a schematic editor, the core data structure of DSP station behavioral synthesis tool suite and a dynamic data-flow simulator. He was an early adopter of object-oriented programming techniques in general and the C++ programming language in particular. In 1996, he joined the Design Technology for Integrated Information and Communication Systems (DESICS) Division of the Interuniversity Micro Electronics Center (IMEC), Heverlee, Belgium, where he did research on C++-based concurrent timed simulation of embedded systems (TIPSY—comparable to but preceding SystemC), automated overhead removal from object-oriented C++ programs, functional parallelization (SPRINT), translation of C++ code to readable C code and C code cleaning for embedded application. He is the author/coauthor of two patent applications and several papers on these subjects.

Kristof Denolf received the M.Eng. degree in electronics from the Katholieke Hogeschool, Brugge-Oostende, Belgium, in 1998, the M.S. degree in electronic system design from Leeds Metropolitan University, Leeds, UK, in 2000, and is currently a Ph.D. candidate at the Techische Universiteit Eindhoven, The Netherlands. He joined the Multimedia (MM) Group of the Nomadic Embedded Systems Division, at the Interuniversity Micro Electronics Centre (IMEC), Leuven, Belgium, in 1998. His main research interests are the cost-efficient design of advanced video processing systems and the end-to-end quality of experience.

Bart Vanhoof received the Electrical Engineering degree from the Katholieke Universiteit, Leuven, Belgium, in 1989. That year he joined the Interuniversity Micro Electronics Center (IMEC). In the Application Group of the VLSI systems and design methodology (VSDM) Division, he demonstrated the CAD developed in other groups in several projects in the field of advanced speech processing, embedded wireless systems, video codecs, and 3D applications. In the context of this work, he authors and coauthors several papers. Vanhoof is Member of the Multimedia (MM) Group since 1996. This group is part of the Application group of the Nomadic Embedded Systems (NES) Division of IMEC. In this group, he applied the SPRINT tool to an MPEG-4 simple profile video codec. Vanhoof's current aim is the development of multimedia applications on embedded multiprocessor systems, with a focus on both the hardware and the software aspects of these realizations. With this design experience, missing links in the design flow will be identified.

Richard Stahl received the Electrical Engineering degree from the Slovak University of Technology in Bratislava, Slovakia, in 2000. He received the Ph.D. degree from the Katholieke Universiteit, Leuven, Belgium, in 2006. In 2000, he joined the Design Technology for Integrated Information and Communication Systems (DESICS) Division of the Interuniversity Micro Electronics Center (IMEC) in Leuven, Belgium, where he did research on automated techniques for exploration of task-level parallelism in object-oriented programs. He is the author/coauthor of several papers on this subject.